A Column Arrangement Algorithm for a Coarse-grained Reconfigurable Architecture

Yuanqing Guo Cornelis Hoede Gerard J.M. Smit
Faculty of EEMCS, University of Twente
P.O. Box 217, 7500AE Enschede, The Netherlands

Abstract

In a coarse-grained reconfigurable architecture, the functions of resources such as Arithmetic Logic Units (ALUs) can be reconfigured. Unlike the programmability of a general purpose processor, the programmability of a coarse-grained reconfigurable architecture is limited. The limitation might be the number of different patterns or the number of different configurations of each ALU. This paper presents a column arrangement algorithm to sort the elements of patterns to reduce the number of configurations of each reconfigurable ALU. The experimental results show that this algorithm leads to nearly optimal results.

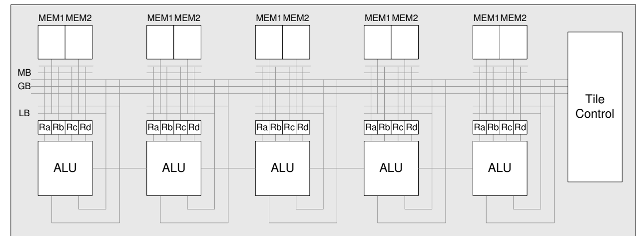

Figure 1. Simplified model of the Montium processor tile

1. Introduction

The most common used computer system architectures in data processing nowadays can be divided into three categories: General Purpose Processors (GPPs), application specific architectures and reconfigurable architectures. GPPs are flexible, but inefficient and have a relatively poor performance. On the other hand, application specific architectures are efficient, show good performance, but are inflexible. Recently reconfigurable systems have drawn more and more attention due to their combination of flexibility and efficiency. Reconfigurable architectures limit their flexibility to a particular algorithm domain. A Montium tile [1] is a coarse-grained reconfigurable system (see Figure 1), designed at the University of Twente, and now commercialized by Recore System [6]. One Montium tile has five ALUs which, for instance, can be configured to compute two additions and three multiplications during the first clock cycle, and one addition, two subtractions and two bit-or operations during the second clock cycle. The combination of concurrent functions that can be performed on the five reconfigurable ALUs in one clock cycle is called *a pattern*.

The programmability of reconfigurable architectures is different from the programmability of GPPs. In a GPP, the function of the ALU can be programmed through instructions. In coarse-grained reconfigurable architectures, the programmability is limited for efficiency reasons. Let us take the control part of the Montium ALUs as an example (see Figure 2). In the Montium, there are 37 control signals for each ALU, so there are 2^{37} possible functions. In practical algorithms only a few combinations are used. The functions that an ALU needs to execute for an algorithm are stored in configuration registers, named *ALU instruction registers* that are located close to the ALU. Totally, there are 8 such ALU instruction registers for each ALU. The contents of the registers are written at configuration time. At runtime, on every clock cycle, one of these 8 ALU instruction registers is selected to control the function of the ALU. An *ALU decoder register*, which is also a configuration register, determines which ALU instruction register is been selected for all five ALUs. A combination of the functions of the five ALUs is called a pattern. As there are 5 ALUs in the Montium, there are 8^5 combinations of the functions for all the ALUs. In other words, an ALU decoder register could have 8^5 different patterns. However, in practice, not all these 8^5 patterns are used for one application. Therefore, there are only 32 ALU decoder registers in the Montium, which can only store 32 distinct patterns. Finally, sequencer instructions will select an ALU decoder register of a particular pattern on every clock cycle. Note

that if there is not enough configuration register space for a specific application, a reconfiguration or partial reconfiguration has to be done. This is very energy-inefficient and time-consuming, and we would like to avoid it. In summary, by using two layers of configuration registers, the control signals for the ALUs in the Montium are reduced from 5×37 to 5 (in the sequencer instruction). On the other hand, the compiler has to the face the challenge of decreasing the number of distinct instructions. It is the compiler's responsibility to consider all these constraints and generate proper configurations at compile time.

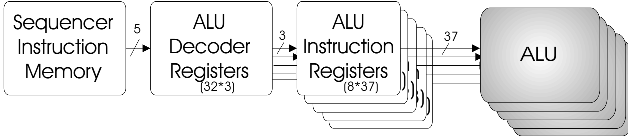

Figure 2. Control part for ALUs in the Montium tile

In compilers, scheduling is used to determine the sequence in which instructions execute. Most scheduling problems are NP-complete problems. To solve the scheduling problems heuristic algorithms have been used to find feasible (possibly suboptimal) solutions. Two commonly used heuristic algorithms are: list scheduling [2][3] and force-directed scheduling [4] [5]. Most of these algorithms assume that a single fixed pattern is used. For instance, in one system, there are two adders and one subtracter. In another system there are two ALUs and one fetch/store unit. In the former one the resources cannot be changed. The function of an ALU in the latter system might change, but for the scheduling problem the model is the same as the former one.

For the Montium, however, this assumption does not hold. The functions of an ALU are not fixed. The final schedule for a Montium tile can be written as a table, called a *schedule table*, of five columns, each column representing the operations of one ALU and each row containing five ALU instructions that will be executed by the five parallel ALUs. The combination of instructions of each row form a pattern. Because the ALU decoder contains at most 32 patterns, the total number of different patterns is 32. All the functions appearing in one column will be executed by one ALU. As mentioned above, there are eight instruction registers per ALU. Therefore, the number of different functions in one column should be smaller than 8. As far as we know, none of the existing scheduling methods take these types of constraints into consideration. In our previous work [7], a multi-pattern list scheduling algorithm is given for the scheduling problem of the Montium, which, assuming a set of patterns are given, schedules a graph in such a way that only one of the given patterns is allowed in each row. The algorithm for selecting a set of patterns is presented in [8]. However, the restriction on the number of different configurations of one ALU is not considered there. As a result, in the final schedule each ALU might end up with more than 8 configurations, which is not a valid schedule. We put all selected patterns into a table, named a *pattern table*. The functions appearing in one column of the pattern table will also be the functions that will appear in the corresponding column of the final schedule table, because the functions of one row in the schedule table must be one row of the pattern table.

In this paper we propose a method to arrange the elements of a set of patterns to minimize the number of different configurations of each ALU.

2. Problem description

In a system with a fixed number (denoted by C, which is 5 in the Montium architecture) of reconfigurable ALUs, each individual ALU can have a limited number of different configurations (the number of different configurations is 8 in the current Montium architecture). We use *a color* to represent the type of a function that a reconfigurable ALU can execute. A pattern might have less than C colors. The undefined elements are represented by dummies (don't care), which means that those elements can be any color. R patterns ($R \leq 32$ in the Montium) can be written as an R by C *pattern matrix*

$$P_{R\times C} = \begin{pmatrix} P_1 \\ P_2 \\ \cdots \\ P_R \end{pmatrix} = \begin{pmatrix} p_{1,1} & p_{1,2} & \cdots & p_{1,C} \\ p_{2,1} & p_{2,2} & \cdots & p_{2,C} \\ \cdots & \cdots & \cdots & \cdots \\ p_{R,1} & p_{R,2} & \cdots & p_{R,C} \end{pmatrix}. \quad (1)$$

In a pattern matrix P the set of different elements appearing in column i defines a set of configurations ALU i will have. This set is called *the configuration set for ALU* i, represented by CS(i, P), $1 \leq i \leq C$.

In this paper, we use X to denote the number of different colors in the pattern matrix defined in Equation (1).

The *concurrence number of a color* x *in a pattern* P_i is denoted by $Con(x, P_i)$, $1 \leq x \leq X$, $1 \leq i \leq R$, which is the number of times the color x is present in pattern P_i. The *maximal concurrence number of a color* x *in a pattern matrix* P is defined as

$$Con_{max}(x, P) = \max_{1\leq i\leq R} Con(x, P_i), \quad 1 \leq x \leq X. \quad (2)$$

For instance, the maximal concurrence number of "a" in the example of Table 1 is two since pattern 1 has two "a"s, the maximal concurrence number of "b" is one.

The column arrangement problem is defined as:
Given a pattern $P_i = \{p_{i,1}, p_{i,2}, \cdots, p_{i,C}\}$, one ordering of its elements is written as $Ord(P_i) = \{p_{i,o(1)}, p_{i,o(2)}, \cdots, p_{i,o(C)}\}$. The goal of the column arrangement algorithm is to find a proper ordering for each row of the pattern matrix P such that the column condition

$$\max_{1 \leq i \leq C} |CS(i, P)| \leq CS_{def}$$

is satisfied, where CS_{def} is the maximum number of allowed different functions of each ALU ($CS_{def} = 8$ for the Montium).

In a real application, the whole application might be scheduled part by part. If we use fewer configurations in one part, we might have more freedom in other parts. Therefore, at the column arrangement phase, we not only aim at satisfying the constraints, i.e., to minimize function defined in Equation (3)

$$f_{max} = \max_{1 \leq i \leq C} |CS(i, P)|, \quad (3)$$

but also aim at minimizing the sum of the size of the configuration sets, i.e., to minimize the function defined in Equation (4)

$$f_{sum} = \sum_{1 \leq i \leq C} |CS(i, P)|. \quad (4)$$

3. Lower bound

Theorem 1 *The sum of the number of column colors in each column cannot be smaller than the sum of maximum concurrence for all colors, i.e.,*

$$f_{sum} \geq \sum_{1 \leq x \leq X} Con_{max}(x). \quad (5)$$

The maximal size of the configuration sets is thus satisfying

$$f_{max} \geq \lceil \frac{1}{C} \sum_{1 \leq x \leq X} Con_{max}(x) \rceil, \quad (6)$$

where $\lceil y \rceil$ *denotes the smallest integer that is larger or equal to y.*

Proof If Equation (5) is not satisfied, there must be a color x which appears a times in the C configuration sets and its maximum concurrence number satisfies $Con_{max}(x) > a$. According to the definition of the maximum concurrence number given by Equation (2), there must be a pattern P_i with $Con_{max}(x)$ times color x. In other words, there must be a row in which x appears $Con_{max}(x)$ times in the matrix defined by Equation (1). This conflicts with the conclusion $Con_{max}(x) > a$ given above.

The Equation (6) can be directly obtained from Equation (5). □

4. Algorithm description

The column arrangement algorithm is given in Figure 3. The algorithm moves patterns from the input pattern matrix Pi to the output pattern matrix Po one by one according to a cost function. The key point is the cost function, which determines which pattern and at which ordering is going to be put to the output matrix at the next step.

//	The input matrix is Pi, and the output matrix is Po;
1.	Take one pattern from Pi as the starting pattern and put it to Po without care of the ordering of the pattern; (We will explain how to select the pattern later.)
2.	For each ordering of each remaining pattern in Pi, compute the cost function.
3.	Choose the pattern and the ordering with the lowest cost and put it to Po.
4.	Repeat step 2 and step 3 until Pi is empty.

Figure 3. A Column Arrangement Algorithm

Cost function To build the cost function used in step 2 and step 3 of Figure 3, the following aspects have to be taken into consideration:

1. It is preferable to put a color to the column which already has the same color because we do not have to add a new color.

2. When we select a column for color x, the column without y is preferred if colors x and y exist together in one pattern. Otherwise, one of them has to appear at least twice in the output configurable sets $CS(i, Po)$. An exception is when $Con(x) > 1$ or $Con(y) > 1$ because a color x has to appear in at least $Con(x)$ configurable sets.

3. The patterns with more dummies have more freedom to arrange, so they are given low priority.

4. For a color x, the pattern P_i is given higher priority to arrange when $Con_{max}(x, P) = Con(x, P_i)$. Then other patterns which have fewer x's can be arranged according to the x's already arranged.

Now we define the conflict function. The intuitive notion of this cost function is based on the above aspects, which will be explained later.

The *conflict factor* between two colors x and y:

$$Conflict(x,y) = \begin{cases} 2000 & \text{if } x \text{ and } y \text{ appear in one pattern and } Con(x)=1, Con(y)=1; \\ 200 & \text{if } x \text{ and } y \text{ appear in one pattern and } Con(x) \geq 2 \text{ or } Con(y) \geq 2; \\ 0 & \text{otherwise.} \end{cases} \quad (7)$$

The *benefit* of assigning one color x to a column i is defined as

$$Benefit(x,i) = \begin{cases} 2000 & \text{if } x \in CS(i,Po) \\ 0 & \text{if } x \notin CS(i,Po) \end{cases}$$

The *size factor* of assigning an element to a column i is:

$$SizeFactor(x,i) = \begin{cases} |CS(i,Po)+1|^2 & \text{if } x \notin CS(i,Po); \\ 0 & \text{if } x \in CS(i,Po). \end{cases} \quad (8)$$

The *cost function of assigning one color x to a column i of a matrix P* is defined as:

$$Cost(x,i) = -Benefit(x,i) + \sum_{y \in CS(i,Po)} Conflict(x,y) + SizeFactor(x,i).$$

For a color x, the pattern P_i is given higher priority to select when $Con_{max}(x,P) = Con(x,P_i)$. Then other patterns which have x's can arrange themselves according to the existing x's. The concurrency factor of pattern P_i is defined as

$$Concurrency(P_i) = \begin{cases} (Con_{max}(x,P))^2 \times 500 & \text{if there is a color } x \text{ with } Con(x,P_i) = Con_{max}(x,P) \text{ and } Con_{max}(x,P) > 1 \\ 0 & otherwise \end{cases}$$

The cost of assigning one pattern P_i to the output pattern matrix Po in the ordering

$$Ord(P_i) = \{p_{i,o(1)}, p_{i,o(2)}, \cdots, p_{i,o(C)}\}$$

is:

$$Cost(Ord(P_i) = \{p_{i,o(1)}, p_{i,o(2)}, \cdots, p_{i,o(C)}\}) = \sum_{1 \leq i \leq C} Cost(p_{i,o(1)}, i) + Concurrency(P_i) + 200 \times \text{ Number of dummies in } P_i.$$

When a pattern has many colors that do not exist in the output pattern matrix, it is more likely to make a bad decision when this pattern is arranged earlier since the output pattern matrix has no information on the new colors. The philosophy in choosing those constants in the functions is to let the new arranged pattern bring in as little as possible information at each iteration. The colors enter the output matrix as slowly as possible.

The ordering of the starting pattern will not decrease the performance of the final result. However, the choice of the starting pattern will significantly influence the speed in which all colors enter the output matrix. In the experiments, we also found that the experimental result is very sensitive to the selection of the starting pattern, especially the value of f_{max}. For the same set of patterns given by Table 1, if we use the pattern 1 as the first pattern, the result will be as shown in Table 3. $f_{sum} = 14$ and $f_{max} = 3$. If we use pattern 7 as the starting pattern, the result will be $f_{sum} = 14$ and $f_{max} = 5$. Observing that the number of patterns is not very large in reality (at most 32 for the Montium), instead of designing an algorithm to find the best starting pattern, we just try all of them, and then take the best one.

Theoretically, the two optimality critera given by Equation (3) and (4) may conflict in some cases. The design of the above cost function gives attention to balancing the size of configuration sets among columns as well as decreasing the total number of all configuration sets. For instance, according to the size factor in Equation (8), the cost of putting a new color to a column with more colors is larger than to a column with fewer colors.

Table 1. Input pattern matrix

pattern	1	2	3	4	5
1	a	a	b	c	d
2	h	i	g	g	f
3	a	f	d	h	*
4	d	i	g	*	*
5	d	b	c	a	e
6	f	g	k	i	l
7	a	k	l	*	*
8	c	f	i	j	d

Example: Now we use the example in Table 1, 2 and 3 to demonstrate the column arrangement algorithm. Here we assume $C = 5$ which is the value in the Montium architecture. The input pattern matrix is shown in Table 1, where the letters represent colors. We use the pattern 1 as the starting pattern. The five configuration sets are now: $CS(1,Po) = \{\text{a}\}, CS(2,Po) = \{\text{a}\}, CS(3,Po) = \{\text{b}\}, CS(4,Po) = \{\text{c}\}, CS(5,Po) = \{\text{d}\}$. When pattern 5 is ordered as {a,e,b,c,d}, column 1, 3, 4 and 5 will have benefit 2000. The conflict factor can be found in table 2. The cost of putting "e" to column 2 is:

Table 2. Conflict factor table

pattern	a	b	c	d	e	f	g	h	i	j	k	l
a	0	200	200	200	200	200	0	200	0	0	200	200
b	200	0	2000	2000	2000	0	0	0	0	0	0	0
c	200	2000	0	2000	2000	2000	0	0	2000	2000	0	0
d	200	2000	2000	0	2000	2000	200	2000	2000	2000	0	0
e	200	2000	2000	2000	0	0	0	0	0	0	0	0
f	200	0	2000	2000	0	0	200	2000	2000	2000	2000	2000
g	0	0	0	200	0	200	0	200	200	0	200	200
h	200	0	0	2000	0	2000	200	0	2000	0	0	0
i	0	0	2000	2000	0	2000	200	2000	0	2000	2000	2000
j	0	0	2000	2000	0	2000	0	0	2000	0	0	0
k	200	0	0	0	0	2000	200	0	2000	0	0	2000
l	200	0	0	0	0	2000	200	0	2000	0	2000	0

$$\begin{aligned} Cost(\text{e}, 2) &= Conflict(\text{a}, \text{e}) + SizeFactor(\text{e}, 2) \\ &= 200 + 2^2 = 204. \end{aligned} \quad (9)$$

The total cost will be $-8000+204 = -7796$. This ordering of the pattern is chosen because all orders and other patterns will have higher cost. The configuration sets are changed to $CS(1, Po) = \{\text{a}\}, CS(2, Po) = \{\text{a,e}\}, CS(3, Po) = \{\text{b}\}, CS(4, Po) = \{\text{c}\}, CS(5, Po) = \{\text{d}\}$. For pattern 8, obviously “c” and “d” will be put in column 4 and column 5 separately. From Table 2 we see that $Conflict(\text{a,f}) = 200$, $Conflict(\text{e,f}) = 0$, $Conflict(\text{b,f}) = 0$. Thus it is better to put “f” in column 3. “i” and “j” can be put in any order.

Table 3. Output pattern matrix: $f_{sum} = 14, f_{max} = 3$

step	pattern	1	2	3	4	5
1	1	a	a	b	c	d
2	5	a	e	b	c	d
3	8	j	i	f	c	d
4	3	a	*	f	h	d
5	2	g	i	f	h	g
6	6	g	i	f	k	l
7	4	g	i	*	*	d
8	7	a	*	*	k	l
CS		3	3	2	3	3

5. Computational complexity

Since one pattern has $C!$ possible orders, if there are k patterns to arrange, step 2 and step 3 in Figure 3 have to choose the best one from $k \times C!$ candidates. If the input pattern matrix has R rows, the number of candidates for arranging the second one is $((R-1) \times C!)$, the number for the third one will be $((R-2) \times C!)$, $\cdots$. Totally, the cost function will be computed $R(R-1)/2 \times C!$ times. Considering that the program given in Figure 3 will be run R times since every pattern can act as a first pattern, the computational complexity of the column arrangement algorithm will be $O(R^2(R-1)/2 \times C!)$. For the Montium application, $R = 32$ and $C = 5$, the algorithm can finish within a couple of seconds on a general Pentium 4 computer.

6. Experiment

Table 4 shows the experimental results for the column arrangement algorithm. The input matrices are randomly generated. We can see the algorithm works very well for minimizing the f_{sum} which is very close to its lower bound. The last column of the table indicates whether the pattern which reaches f_{max} also reaches f_{sum}. In our experiments, the answer is almost always “yes”. The lower bound 1 of f_{max} in the table is defined by Equation (6). The lower bound 2 is computed by $\lceil f_{sum}/\text{C} \rceil$. The lower bound 2 indicates how good the algorithm balances the size of all configuration sets. In the experiment, f_{max} mostly reaches the lower bound2.

7. Conclusion

This paper presents a column arrangement algorithm, which is used to decrease the number of configurations for each reconfigurable ALU. This algorithm uses a heuristic approach to decrease the computational complexity. To diminish the negative effect caused by the greedy character of heuristic methods, some “control” functions (cost functions) are defined which try to tackle the problem in a more sophisticated way. From the experimental results we can see that the functions lead to very good results. The algorithm decreases the maximum number of configurations of each ALU and the total number of the configurations of all ALUs at the same time. The latter is useful when the scheduling problem of a big application is tackled part by part.

Table 4. Experimental result

Number of patterns	Number of colors	Lower bound of f_{sum}	f_{sum}	Lower bound1 of f_{max}	Lower bound2 of f_{max}	f_{max}	Simultaneously?
10	10	15	16	3	4	4	Yes
10	10	14	16	3	4	4	Yes
10	9	19	19	4	4	4	Yes
10	10	14	15	3	3	3	Yes
10	9	15	15	3	3	3	Yes
10	8	14	14	3	3	4	No
10	8	14	14	3	3	4	Yes
10	6	13	13	3	3	3	Yes
10	6	12	12	3	3	3	Yes
10	12	18	18	4	4	4	Yes
20	20	29	30	6	6	7	Yes
20	20	29	33	6	7	8	Yes
20	25	31	36	7	8	8	Yes
20	23	27	29	5	6	7	Yes
32	10	22	24	5	5	5	Yes

8. Acknowledgments

This research is conducted within the Chameleon project (TES.5004) and Gecko project (612.064.103) supported by the PROGram for Research on Embedded Systems & Software (PROGRESS) of the Dutch organization for Scientific Research NWO, the Dutch Ministry of Economic Affairs and the technology foundation STW.

References

[1] Paul M. Heysters, Gerard J.M. Smit, E. Molenkamp: “A Flexible and Energy-Efficient Coarse-Grained Reconfigurable Architecture for Mobile Systems”, *The Journal of Supercomputing*, Vol 26, No. 3, Kluwer Academic Publishers, Boston, U.S.A., November 2003, ISSN 0920-8542.

[2] B.M. Pangrle and D.D. Gajski, “Design Tools for Intellegent Compilation,” *IEEE Trans. Computer-Aided Design,* Vol. CAD-6, No. 6, Nov.1987, pp. 1098-1112.

[3] T.C. Hu, “Parallel Sequencing and Assembly Line Problems,” *Operations Research,* Vol.9, No.6, Nov.1961. pp. 841-848.

[4] P.G. Paulin and J.P. Knight, “Algorithms for High-Level Synthesis,” *IEEE Design and Test of Computers,* Vol.6, No.4, Dec. 1989, pp.18-31.

[5] P.G. Paulin and J.P. Knight, “Force- Directed scheduling for behavioral synthesis of ASIC’s”, *IEEE Trans. Computer-Aided Design*, vol. 8, pp. 661-679,March 1989.

[6] http://www.recoresystems.com.

[7] Yuanqing Guo, Cornelis Hoede, and Gerard J.M. Smit, “A Multi-Pattern Scheduling Algorithm”, in *Proceedings of ERSA 2005*, June 27-30, 2005, Monte Carlo Resort, Las Vegas, Nevada, USA.

[8] Yuanqing Guo, Cornelis Hoede and Gerard J.M. Smit, “A Pattern Selection Algorithm for Multi-Pattern Scheduling”, Accepted for the 13th Reconfigurable Architectures Workshop (RAW2006), Greece.